

19

Blockchain Security

Other than scalability and privacy, there is another challenge that needs to be addressed in order for blockchains to become even more trustworthy. The issue is that of security. The security of blockchain is crucial because it is the backbone of many use cases, including sensitive ones like healthcare, finance, and supply chain management. With the advent of DeFi, it has become even more important to ensure the correct functioning of blockchain so that there are no vulnerabilities left in the blockchain that could be exploited, resulting in a loss of funds or other unintended consequences.

As the blockchain ecosystem is composed of several layers, there are security concerns at every layer, and we'll discuss them in this chapter. There are also layer 2 (side chains, etc.) related security concerns, which we'll also shed some light on in this chapter.

We'll cover the following topics in this chapter:

- Security in blockchain
- Background and historic attacks
- Blockchain layered model
- Threats and vulnerabilities at each layer of blockchain, including smart contract security, blockchain layer security, and security at other layers, how to address them, and best practices
- Layer 2 security concerns
- Tools and techniques to find vulnerabilities
- Models to perform threat analysis

So, let's begin by first looking at the subject of security in the context of blockchains.

Security

Blockchains are generally secure and make use of asymmetric and symmetric cryptography, as required throughout the blockchain network, to ensure that the core layer of the blockchain is secure. However, there are still a few caveats that can result in the security of the blockchain being compromised.

There have been many high-profile successful attacks on the blockchain ecosystem over the years. Some were outright malicious, and some were accidental; however, they all resulted in significant losses. Some examples include:

- The attack on Binance Smart Chain, which occurred on October 6, 2022. The hackers infiltrated the BNB cross-chain bridge Token Hub where they were able to mint fraudulent BNB tokens, through fake withdrawal proofs. Initially, the estimated damage was 566 million USD.
- In August 2021, the Poly Network theft, which resulted in 610 million USD being stolen. This attack exploited a vulnerability that occurred due to mismanaged access rights between Poly smart contracts.
- In February 2022, the Wormhole bridge exploit, which resulted in 321 million USD being lost. It occurred in February 2022, when a hacker found a vulnerability in the smart contracts that allowed him to mint 120,000 WETH on Solana not backed by collateral and was then able to swap this for ETH.
- In March 2022, Ronin Bridge, which was exploited for around 612 million USD. The reason for this exploit was unauthorized access to private keys, which resulted in compromised validator nodes, which consequently approved fraudulent transactions, draining funds from the bridge.
- The Luna collapse in May 2022, which wiped out an estimated 60 billion USD from the cryptocurrency world. This mainly occurred due to an unfitting algorithm behind the stable coin Luna where UST was backed by Luna, instead of a fiat currency.

Other hacks include MtGox, KuCoin, Pancake Bunny, Parity wallet hack, and many more. There have also been losses incurred due to poor management and controls, such as:

- **FTX collapse:** FTX was one of the biggest cryptocurrency exchanges in the world. It collapsed in November 2022 after a surge of customer withdrawals that FTX failed to fulfill due to not having enough assets in reserve to meet customer demands. Moreover, the situation was exacerbated due to FTX wallets being targeted and hacked by attackers, which resulted in almost 640 million USD being stolen. This was allegedly an insider attack by employees or the result of a malware attack. Eventually, FTX filed for bankruptcy due to liquidity crunches. The collapse of FTX left more than 1 million customers unable to withdraw assets worth an estimated 8 billion USD.
- **Binance client money issue:** In January 2023, Binance mistakenly mixed Crypto.com Exchange's customer funds with B-Token collateral. Thankfully, this mistake was realized before any material damage was done, but it highlights the importance of controls and compliance requirements. This issue, if not spotted earlier, could have led to owners not being able to withdraw money due to a lack of liquidity/funds by the exchange, as there is no segregation of assets between clients' money and any collateral used. This situation would have meant substantial reputational and financial loss due to a lack of liquidity if left unnoticed.

Now, before discussing security further in blockchains, let's have a look at a layered view of blockchain, which helps us to categorize different attacks efficiently. Generally, a blockchain ecosystem is composed of several layers and each layer presents its own security challenges. We can visualize a layered model in *Figure 19.1*:

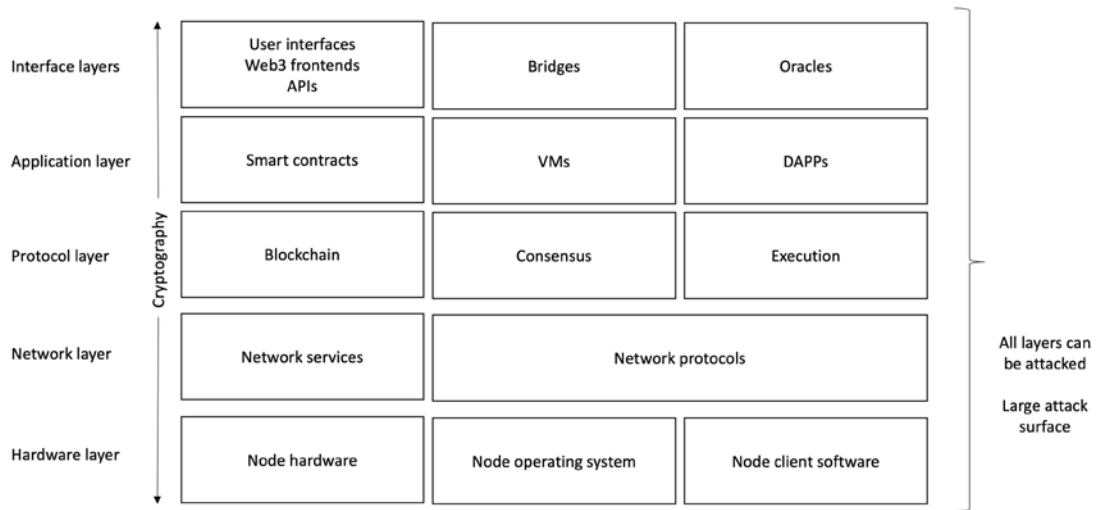


Figure 19.1: Blockchain layered view

In the preceding diagram, note that all layers can be attacked by an adversary and as such, the attack surface is quite large. In other words, it means that the attack surface spans across multiple layers, and an attacker can try to attack all layers, including the hardware, network, protocol, applications, and interface layers. Let's now see what each layer is composed of. Note that an attacker can try to find vulnerabilities in each layer and every component thereof. Also, as the financial gain from these attacks can be significant, hackers could be motivated to attack and try different ways to exploit any possible vulnerability.

Blockchain layers and attacks

In this section, we'll look at how we can see blockchain as a layered architecture to study security issues, attacks, and mitigation.

- **Hardware layer:** This is the core hardware on which a blockchain network node runs.
- **Network layer:** This is the core network layer on which blockchain protocols run. This layer includes protocols like TCP/IP, P2P, UDP, and other relevant services and protocols.
- **Blockchain protocol layer:** This is the blockchain layer where consensus, transaction execution, incentive management, and mining processes run.
- **Blockchain application layer:** This is the layer that consists of smart contracts and virtual machines, e.g., EVMs for executing smart contracts. DAPPs also exist at this layer, and so on.
- **Interface layer:** This layer includes bridges, oracles, governance, user interfaces for Web3 DAPPs, and APIs. While these components could possibly exist in the blockchain application layer, I think differentiating between these components here as a separate layer helps to define and address specific challenges better.
- **Cryptography layer:** Cryptography plays a vital role at each layer. Here, we will treat this as a separate layer, just for the purpose of discussion; otherwise, it exists in some shape or form at every layer in the blockchain ecosystem.

Each layer is open to an adversary to try and exploit any weaknesses or limitations. Blockchain is particularly interesting because while conventional software, hardware, and general information security concerns apply, because it uses conventional hardware and software to run, it also has its own challenges, such as smart contract security. The challenge is protecting the blockchain ecosystem at all layers, especially at the blockchain protocol and blockchain application layers.

In summary, we can say that there is a lot that needs to be secured in a blockchain ecosystem, including layer 1, layer 2, smart contracts, virtual machines, wallets, and even people. Why people? Because they can be victims of social engineering attacks. Moreover, people need to be educated about these threats so that they can identify social engineering patterns and protect themselves.

Cryptography plays an important role at every layer; therefore, we'll discuss cryptography as a separate layer/topic.

Now let's discuss attack vectors and mitigations at each layer.

Hardware layer

As this layer is composed of hardware that runs the operating system and node software, this layer is impacted by the usual threats, including viruses, malware, and unauthorized access. Blockchain-specific malware also exists, which specifically targets cryptocurrency blockchain network nodes. Such crypto malware is especially targeted to perform cryptojacking. Cryptojacking can be defined as the unauthorized use of computing resources that belong to someone to mine cryptocurrency.

Cryptojacking is a method of taking over a computer or web browser to mine for cryptocurrency without the user's permission. A successful cryptojacking attack doesn't necessarily result in software being installed on the victim's computer, but this can work within web browsers directly – so-called “in-browser mining.” This occurs when a user browses to a website that executes a mining script on their browser. Even when the user leaves the website, the mining script can remain executing.

The malware used for cryptojacking is called **crypto malware**. It can infect operating systems including Windows, Linux, macOS, iOS, and Android. Browser extensions sometimes also include JavaScript to mine cryptocurrency, e.g., Archive Poster and Iridium. Due to cryptojacking, the machine performance is significantly reduced as CPUs and GPUs are running the mining process instead of performing the usual expected operations on the system. As we know, mining is a resource-intensive process, especially PoW, so it can be beneficial for hackers to hack into someone else's computer and run mining from their computer.

There can also be a **Denial of Service (DoS)** attack on a blockchain node, which can cause the resources on the computer to struggle to keep up with the incoming requests. In a DoS attack, the attacker floods the node with a large number of requests, overwhelming it and preventing it from processing legitimate transactions.

There can also be remote execution attacks via open ports, not only on the operating system but also in the node software, e.g., Bitcoin nodes' RPC port 8332, which hackers can try to exploit. A list of vulnerabilities is available here for Bitcoin and Ethereum:

- Bitcoin: https://www.cvedetails.com/vulnerability-list/vendor_id-12094/Bitcoin.html
- Ethereum: https://www.cvedetails.com/vulnerability-list/vendor_id-17524/Ethereum.html

There may be other vulnerabilities that have not been discovered yet or could be introduced with future upgrades.

There is also a possibility that the operating system is outdated or even that the node software is outdated, which could open doors for exploitation by hackers. Also, misconfigured operating system or client node software could lead to a weakness that hackers can exploit. The hardware on which the node is running could also be vulnerable due to flaws such as Spectre, Meltdown, and Thunderclap. The firmware can also be exploited due to weaknesses such as Thunderstrike, ROCA, etc. It is important that standard precautionary and security measures such as system hardening, operating system upgrades, node software upgrades, and antivirus software are applied to the hardware node hosting the blockchain client software.

Blockchain node software can be attacked by traditional and blockchain-specific malware, which can result in the theft of private keys, the creation of rogue transactions, the censoring of some traffic, and resource starvation, resulting in a DoS.

Network layer

There can be several attacks that can be carried out on the network layer. We will discuss them next:

1. **Sybil attack:** First, we introduce the Sybil attack. A Sybil attack is a type of network attack in which an attacker creates and uses multiple fake identities, or “Sybil nodes,” to gain an unfair advantage or perform malicious actions within a network. This can be accomplished by creating multiple fake identities and using them to gain a disproportionate level of influence within the network, such as by participating in a consensus process or voting system. The goal of a Sybil attack is to manipulate the network or disrupt its normal operation, often for personal or financial gain. If enough fake identities are created, this attack can be used to outvote the honest nodes on the network. These nodes can then take over the network and refuse to receive or transmit a block, effectively rendering the network useless. If attackers manage to control more than 51% of the hash rate, then they can reorder the transactions, censor transactions, and effectively take over the entire network. Consensus protocols are designed in such a way that Sybil attacks are not possible, though in some cases, due to weaknesses and bad design, Sybil attacks can still work.
2. **DoS attack:** The network can also be impacted by DoS, which could result in overwhelming the network with unnecessary traffic to make it inaccessible to users. While blockchain is a decentralized peer-to-peer network with SPOF, these networks are generally immune to network DoS attacks. However, some points of centralization exist in most blockchains, which can be targeted in a DoS attack; for example, boot nodes can be attacked, which are known and hardcoded in the blockchain client node software. Of course, here traditional network protection mechanisms to protect against DoS can be used, such as firewalls, intrusion detection and prevention systems, load balancers, rate limiters, and traffic shaping techniques.
3. **Eclipse attack:** An eclipse attack is a type of attack on a blockchain network in which an attacker tries to isolate a specific node or some nodes from the rest of the network, effectively “eclipsing” them and making them unable to communicate with other nodes, stopping them from receiving new information.

In this attack, a malicious node can control routes of communication between two segments of the blockchain network, which allows this node to control the information flow between these two network segments. This node can stop forwarding transactions and blocks to a victim node, resulting in it being “eclipsed.” An eclipse attack can result in DoS because the victim node cannot receive and process any new transactions. It can also result in double-spending because if the targeted node is not able to communicate with the rest of the network, it is not aware of transactions that have been broadcasted and confirmed by other nodes. This can allow the attacker to double-spend their digital assets because the targeted node will not have the most up-to-date information about the blockchain.

4. **Network spoofing:** Another attack could be network spoofing. In blockchain networks like Ethereum and Bitcoin, boot nodes are hardcoded in the node client software and are used to provide a starting point for new nodes joining the network to start searching for neighbor nodes. It is possible for an attacker to launch a “spoofing” attack against a boot node by impersonating the boot node. New nodes will connect normally to this false boot node, which could result in a denial of service attack and other security implications.

Blockchain layer

The blockchain layer represents the layer 1 or layer 2 blockchain that is the core of the protocol. There are several attacks that can be carried out here.

Attacks on transactions

There are several attacks that can be carried out on transactions. The transactions are constructed by the users, and they can be malformed or invalid. As transactions are propagated to all nodes and all nodes process all transactions (except light nodes), rogue transactions or transactions especially crafted to cause harm can become an attractive attack vector for hackers. The transaction can contain a peculiar data structure, which can cause nodes to malfunction. Usually, this happens due to a bug in the node software, but theoretically, it is possible for a virus or malicious code to be spread through transactions in a blockchain network.

While invalid and incorrectly signed transactions with invalid signatures will be rejected by all nodes, it is possible that the transaction is properly signed and passes all checks before executing transactions and ending up executing some rogue code that results in an undesirable effect. One example is from 2010 when over 184 billion Bitcoins were created out of thin air due to an integer overflow vulnerability in the Bitcoin core node software. Some relevant attacks are transaction malleability, which allows a hacker to modify the unlock script of a pending transaction and still pass the transaction validation checks.

This attack would allow changing the transaction ID of the transaction, which could lead to a situation where the recipient of the transaction could claim that they never received the funds, tricking the sender to send the payment again. The Bitcoin core software is now quite stable and has gone through several bug fixes, upgrades, and rigorous testing, such as a segregated witness upgrade, which fixed the transaction malleability issue. Therefore, such attacks are quite difficult to achieve on a Bitcoin network. However, other cryptocurrency blockchains could have some zero-day vulnerabilities that can be exploited through malformed transactions. Even Bitcoin could still have some unknown vulnerabilities.

The serialization and deserialization of data for transactions and blocks is a standard operation that blockchain nodes do. It is possible that if some data structure contains malicious code, when it is deserialized by a node, it could result in integer overflow, buffer overflow issues, or even DoS. Nodes could end up executing malicious code if they are running old software that contains a specific bug related to this vulnerability where after being deserialized, the code doesn't check for all conditions.

It is also possible that due to the ability to encode some arbitrary data in a transaction, this can be achieved. For example, through a Bitcoin transaction's Coinbase field, the `OP_RETURN` opcode, or some other relevant technique. With this ability to embed arbitrary data in a transaction, it is possible for a special transaction to be crafted that, once executed, could result in the malfunction of not only nodes but also other software that reads transactions, such as Blockchain Explorer, third-party monitoring tools, and some enterprise backend software reading transactions from a private blockchain or even a public blockchain.

As blockchains have other surrounding software, including blockchain explorers, frontends for DAPPs, and other user frontends, it is possible, by using injection attacks, for the blockchain to be attacked. For example, inadequate input validation, cross-site scripting attacks, or inappropriate inputs, e.g., asking the user to enter their private keys in a web frontend, could lead to private key exposure and other unintended consequences, such as loss of funds, unintended execution of smart contracts, and rogue transaction execution or a DoS attack. Similar attacks can be performed at the block level too.

Transaction replay attacks

A replay attack is a type of vulnerability that arises when two separate cryptocurrencies based on a fork of the same original chain allow transactions to be recognized as valid on both chains.

In a transaction replay attack, an attacker captures a valid transaction from a network and tries to reuse it by “replaying” it on the same or a different network. This can be done by an attacker who has gained access to the transaction data, for example, by intercepting it over the network or by gaining access to a device that was used to initiate the transaction. If the transaction is replayed on a different network, it can potentially be used to transfer funds or other assets to the attacker's own account.

Replay attacks can be a problem in networks that use the same keys for signing transactions on different networks, or that do not have a mechanism in place to prevent the reuse of transactions. To protect against replay attacks, it is important to use unique keys to sign transactions on different networks and to implement a mechanism for detecting and preventing the reuse of transactions.

In 2016, Ethereum suffered from a transaction replay attack when the Ethereum network underwent a hard fork to resolve the “DAO hack” issue. A hard fork is a change to the protocol of a blockchain network that is not backward compatible, meaning that all nodes on the network must upgrade to the new version of the protocol or else they will be unable to participate in the network.

During the hard fork, the Ethereum network split into two separate networks: **Ethereum (ETH)** and **Ethereum Classic (ETC)**. This meant that transactions on one network would not be recognized on the other network, and vice versa. However, because both networks were using the same keys to sign transactions, an attacker was able to capture a valid transaction on one network and replay it on the other network.

To fix this issue, Ethereum implemented a new transaction signing algorithm implemented under **EIP-155** that included a chain ID in the signed data. This chain ID is unique to each network and allows the network to verify that a transaction was intended for it and not another network. As a result, transaction replay attacks are no longer possible on Ethereum.

Recently, replay attacks have come into the limelight again after the merge (the Ethereum PoW to **Proof of Stake (PoS)** switch) in September 2022. In order to prevent replay attacks, the chain ID of the old forked-off Ethereum PoW chain was changed from 1 to 10,001. While this change protects against core chain replay attacks, it doesn't protect against some loopholes that emerged as a result of not updating smart contracts, DAPPs, and other peripheral components around the core chain. A lot of contracts remain vulnerable to replay attacks.

One key example is an attack on Omni Bridge smart contracts. This replay attack launched against the Ethereum old PoW chain and Omni Bridge resulted in the loss of 200 WETH from the Ethereum PoW chain. The attacker first transferred 200 WETH from the Ethereum new PoS chain (after the merge) and was replayed on the Ethereum PoW chain. Omni Bridge did not validate the chain ID before approving the transaction, which resulted in the PoW chain losing 200 WETH and attackers earning an extra 200 WETH. One thing to note is that this wasn't a true replay attack but a *calldata* replay; however, it highlights the importance of ensuring that whenever there are upgrades, and other changes in the core protocol, protection against replay attacks is baked into the new protocols. The transaction wasn't a replay attack on the chain, but a *calldata* replay attack, which occurred due to a flaw in the specific contract on the Gnosis chain. This raises a general concern about peripheral systems, especially bridges, which must verify the correct chain ID of the cross-chain messages. This can be seen as an attack at the interface layer too, which we'll discuss shortly.

Blockchain bridges have become a target for hackers. More on this shortly, when we discuss the interface layer.

Attacks on consensus protocols

There are several attacks that can be carried out against consensus protocols in blockchain, including:

1. **A 51% attack:** This occurs when a miner or group of miners controls more than 50% of the mining power on the network, allowing them to control the confirmation of transactions and potentially double-spend coins.
2. **Selfish mining:** This occurs when a miner withholds the blocks they mine from the network in order to increase their chances of finding the next block and earning the block reward.
3. **An eclipse attack:** This occurs when an attacker is able to isolate a node from the rest of the network, allowing them to control the information the node receives and potentially manipulate the consensus process.
4. **A Sybil attack:** This occurs when an attacker creates multiple identities or "sybils" in order to control a significant portion of the network and influence the consensus process.
5. **A nothing-at-stake attack:** This occurs in PoS-based blockchain where validators are not required to put up any collateral, allowing them to vote on multiple chains and potentially undermine the integrity of the network.

There are several attacks against the PoS mechanism:

1. **Nothing-at-stake problem:** This attack occurs when there are only rewards in a PoS mechanism, and no penalties. This lax situation can result in a validator attesting any blocks and/or multiple forks/blocks of a blockchain to increase rewards for them.
2. **Stake grinding attack:** In this attack, an attacker tries to bias the validator selection algorithm to favor the selection of their own validators. This would result in the attacker's validator being selected more than their fair share and would result in minting more unjustified rewards.

Double-spending

In a double-spending attack, a malicious user sends the same cryptocurrency or token to two or more different recipients, effectively spending the same funds twice. There are different types of attacks that can result in double-spending. These attacks include a race attack, a Finney attack, and a 51% attack.

A race attack is a type of double-spending attack that occurs when a fast transaction is broadcasted before a slow transaction that spends the same coins.

In a Finney attack, a malicious actor first sends a transaction to a merchant for a certain amount of cryptocurrency and then quickly mines a new block that doesn't include this transaction. The attacker then sends a second transaction spending the same coins to a different recipient before the first transaction can be confirmed by the network. Since the attacker can mine the new block and validate the second transaction before the first transaction can be confirmed, the merchant is left with an invalid transaction and no payment. Finney attacks are low risk since they require the attacker to have significant computational power to mine a new block before the first transaction can be confirmed. However, they are still a potential threat and should be taken into consideration when designing and implementing cryptocurrency systems.

Selfish mining

A selfish mining attack can occur when an attacker manages to solve the PoW and mine a block but instead of announcing it to the network, they withhold it from the public chain. This would create a fork, and the attacker can then keep mining on the new forked chain, albeit only known to them, to build an alternative chain and get ahead of the public blockchain. When this malicious chain is ahead of the honest public chain, the latest block can be released to the network. As the network is programmed to accept the most recent block, the malicious fork will overwrite the honest/original chain. This attack can result in achieving virtually any type of malicious goal, including double-spending, stealing cryptocurrency, censoring transactions, and similar goals.

Forking and chain reorganization

Forking in blockchain refers to the creation of two or more parallel chains from a single blockchain. This can occur when multiple validators on the network generate blocks simultaneously, leading to two or more separate chains of blocks that are different from each other. Chain reorganization refers to the process of changing the current longest chain in a blockchain network. This can occur if a longer chain becomes available after a network split or if a malicious miner creates an alternative, longer chain.

An attacker can induce these attacks by using various methods, such as the nothing-at-stake attack, the long-range attack, or the selfish mining attack. In such attacks, a malicious validator may attempt to manipulate the blockchain by creating multiple chains or altering the longest chain, with the goal of compromising the integrity and security of the network.

Blockchain application layer

There are several vulnerabilities that can exist at this layer. The most prominent are smart contract-related vulnerabilities. What makes smart contract security particularly interesting is that there is no patching after deployment, which is different from traditional software. A proxy pattern can help here, but remember it is best to avoid bugs.

Smart contract vulnerabilities

Several security bugs in smart contracts have been discovered and analyzed in the wild. These include transaction ordering dependence, timestamp dependence, mishandled exceptions such as call stack depth limit exploitation, and reentrance vulnerability. Attacks can be classified into malicious actions, improper design (protocol or app), bugs, and social engineering. Now we present a list of smart contract vulnerabilities:

1. **The transaction ordering dependency bug** basically exploits scenarios where the perceived state of a contract might not be what the state of the contract changes to after execution. This weakness is a type of race condition. It is also called front-running and is possible due to the fact that the order of transactions within a block can be manipulated. As all transactions first appear in the memory pool, the transactions there can be monitored before they are included in the block. This allows a transaction to be submitted before another transaction, thus leading to controlling the behavior of a smart contract.
2. **Timestamp dependency bugs** are possible in scenarios where the timestamp of the block is used as a source of some decision-making within the contract, but timestamps can be manipulated by the miners (block producers). The *call stack depth limit* is another bug that can be exploited due to the fact that the maximum call stack depth of EVM is 1,024 frames. If the stack depth is reached while the contract is executing, then in certain scenarios, the send or call instruction can fail, resulting in the non-payment of funds. The call stack depth bug was addressed in the EIP-50 hard fork at <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md>.
3. **Send fail issue:** When sending funds to another contract, sending can fail, and even if throw is used as a *catch-all* mechanism, it will not work.
4. **Timestamp dependence** is another vulnerability that is quite common. Usually, the timestamp of a block is accessed via `now` or `block.timestamp`, but this timestamp can be manipulated by miners, leading to influencing the outcome of a function that relies on timestamps. Often, this is used as a source of randomness in lottery games to select the next winner. Thus, it might be possible for a miner to modify the timestamp in such a way that the chances of becoming the next winner increase.
5. Other usual software bugs, such as **integer overflow and underflow**, are also quite significant, and any use of integer variables should be carefully implemented in **Solidity**.

For example, a simple program where `uint8` is used to parse through elements of an array with more than 255 elements can result in an endless loop. This occurs because `uint8` is limited to 256 numbers.

6. **Reentrancy** is a common bug in smart contracts. The reentrancy bug was exploited in the DAO attack to siphon out millions of dollars into a child DAO. The reentrancy bug essentially means that a function can be called repeatedly before the previous (first) invocation of the function has completed. In other words, reentrancy attack occurs when an attacker controlled contract manages to call back into a smart contract before the first function call is complete.

This is particularly unsafe in Ether withdrawal functions in Solidity smart contracts.

7. **Unguarded selfdestruct**: If there are no proper access controls implemented, anyone can destroy the contract by calling self-destruct. This famously occurred in Parity wallet self-destruction where a user accidentally triggered the self-destruct function in Parity smart contract code, which resulted in a loss of 513,774.16 Ether. Even though it was based on multi-signature and seemed to be a secure contract, the self-destruct function was not secured, so when called, it resulted in the deletion of the code.
8. **Unprivileged write to storage**: This attack occurs when a smart contract does not properly validate and restrict access to its state variables.
9. **Integer underflow and overflow** occur when an operation is performed on an integer that results in a value outside the range of the integer data type. For example, in a 256-bit unsigned integer, the minimum value is 0, and the maximum value is $2^{256}-1$. If an operation subtracts 1 from 0, it results in an underflow, and if an operation adds 1 to $2^{256}-1$, it results in an overflow. An attacker can use integer underflow and overflow to manipulate a contract's state or cause it to behave unexpectedly. For example, they could use an underflow to change the balance of an account to a large positive value or use an overflow to make a value appear to be 0. To prevent integer underflow and overflow in Solidity, developers can use libraries such as SafeMath, which automatically checks for underflow and overflow and reverts the transaction if it occurs. Additionally, developers can use specific data types, such as `uint256`, to limit the range of values an integer can take and write tests to validate the contract's behavior under different inputs.
10. **Oracle manipulation**: An oracle manipulation attack in blockchain occurs when an attacker modifies the data being provided to a smart contract by an oracle. This can be done by either compromising the oracle itself or by exploiting vulnerabilities in the smart contract that integrates with the oracle. These attacks can occur due to various vulnerabilities, such as:
 1. **Lack of data validation**: If the smart contract does not validate the data received from the oracle, an attacker can provide false or malicious data to the contract.
 2. **Unrestricted access**: If the oracle is accessible by anyone, an attacker can modify the data being sent to the contract.
 3. **Poorly designed access control**: If the smart contract does not have proper access controls in place, an attacker can modify the oracle to provide malicious data to the contract.

To prevent oracle manipulation attacks in blockchain, data validation, appropriate access control, and digital signatures can be used. In addition, a multi-oracle approach can be used where multiple oracles are used to provide the same data to the contract, which can reduce the risk of manipulation. If one oracle is compromised, the contract can still receive accurate data from the other sources:

1. **tx.origin:** In Solidity, `tx.origin` is a global variable that returns the address of the account that sent the transaction. Using this variable for authorization could render a smart contract vulnerable if an authorized account calls into a malicious contract. A call made to the vulnerable contract would pass the authorization check because `tx.origin` returns the original sender of the transaction, which in this case is the authorized account. In order to remediate this issue, do not use `tx.origin` for authorization; instead, use `msg.sender`.
2. **Transaction order dependence:** A transaction order dependence vulnerability can occur when the execution of a smart contract depends on the order in which transactions are processed on the blockchain. This can create a race condition where the outcome of a contract can be altered based on the order in which transactions are processed, potentially allowing attackers to manipulate the intended results. For example, a smart contract might have a function that rewards the first person to submit a solution to a problem. If two people submit solutions, one with a standard gas fee and another (attacker) with a higher gas fee, then the miner who confirms the block will choose the transaction with the highest gas price to be included first, resulting in awarding the prize to the person who paid a higher gas fee, instead of the person who solved the problem first. This vulnerability can also occur in ERC-20 contracts where the approve function allows one address to allow another address to spend tokens on its behalf. An attacker who is aware of a change in approval can submit a transfer request with a higher gas price, allowing them to receive more tokens than intended.
3. **Bad randomness:** Randomness is useful in many applications, for example, gaming apps or lottery apps, where a random winner is chosen based on the generated random number. Strong randomness is, however, not easy to generate. Usual sources like `block.timestamp` are insecure as a miner can choose to provide any timestamp within a few seconds and still have their block accepted by others. `blockhash` and `block.difficulty` are also insecure as miners have control over them. In order to remediate this issue, it's recommended to use an externally trusted source of randomness through multiple oracles or use a RANDAO.
4. **Using an outdated compiler:** Using an outdated Solidity compiler version could lead to issues, especially if there are publicly disclosed bugs that adversely affect the current compiler version. It is recommended to use a recent version of the Solidity compiler to avoid such issues.
5. **Unchecked call return value:** The message call return value is not checked. Execution will continue even if the called smart contract throws an error. If the message call fails accidentally or a hacker somehow manages to induce a failure on the call, such a situation can result in unexpected behavior in the program. It is recommended to check the return value of the call to handle the call failure. Also, sometimes it is recommended to avoid making external calls if possible. It might be possible for your contract to be manipulated by a third party/external malicious contract. This occurs because the execution control is transferred to an external contract during an external call. Attacks such as reentrancy and race conditions could occur due to this.

DeFi attacks

1. **Flash loan attacks:** In a flash loan attack, an attacker can borrow large amounts of funds with no collateral. This way, the attacker manages to move the market in their favor; e.g., they manipulate the price of a cryptocurrency asset on one exchange, gaining funds and then immediately selling it on another.
2. **Sandwich attacks:** A “sandwich attack” is a type of cryptocurrency front-running scheme that involves the attacker scanning for high-value transactions on the blockchain (in mempools), then placing a transaction ahead of the victim’s with a higher gas fee to ensure it is processed first. This causes the price of the coin to increase, making the victim pay more. The attacker then sells their own coins, profiting from the difference in price. This tactic is referred to as a “sandwich attack” because it sandwiches the victim’s transaction between two transactions submitted by the attacker. Over the period of May 2020 to April 2022, it was estimated that over 60,000 Ether, worth over 72 million USD, was lost due to these types of attacks.
3. **MEV/BEV:** MEV refers to the highest amount of value that can be gained beyond the regular block reward and gas fees through the manipulation of transactions within a block, such as by including, omitting, or altering the order of transactions.
4. **Stablecoins stability/security risks:** There can be several risks for stable coins, including liquidity risk, solvency risk, and regulatory risk. Stable coins are usually backed by reserve assets but if the demand for the stable coin increases, there may not be enough reserve assets to meet the demand, leading to a drop in its value.
5. **Identity spoofing:** In the DeFi ecosystem, identity and access control is of the utmost importance. It must be ensured that only authorized entities can operate on the network. This can be achieved by verifiable credentials and DIDs. We will discuss this in more detail in *Chapter 20, Decentralized Identity*. The key problem here is how to verify that an identity is true.
6. **Forged NFTs:** This is where a scammer creates art or even generates art online and claims it is by a famous artist. Also, in some cases, the scammer may take original art from an artist, copy it, create an NFT out of it, and sell it with their name. In both these situations, the original artist does not get their fair payment.
7. **NFT DoS:** This can occur due to the off-chain storage of actual NFT art where an attacker takes over or takes down the server where the actual art is hosted.
8. **Unlimited (scarcity-free) token generation:** There is a possibility that tokens are generated without any economic scarcity, which results in the devaluation of the asset.
9. **Airdrop hunting:** Usually, airdrops are limited in supply and a fixed number is intended for a single user; however, it might be possible for a scammer to create multiple identities (usually just multiple emails) and claim more airdrops than intended by the airdrop provider.

Next, we will discuss the interface layer.

Interface layer

Several attacks exist at the interface layer, as described below.

Oracle attacks/oracle manipulation attacks

The utilization of blockchain oracles poses a potential security risk due to the dependence on external information they provide. These oracles, typically implemented as smart contracts, have the capability to supply incorrect data, which can result in detrimental consequences for processes linked to the data feed. Such manipulation can have far-reaching effects, including unwarranted asset liquidations and malicious trading practices. In this context, it is crucial to examine and understand the common vulnerabilities and failures associated with blockchain oracles. Some oracle attacks include:

- **Tampering with data sources:** Attackers can manipulate the data sources that the oracles rely on to feed false information into the blockchain, potentially causing the smart contract to make incorrect decisions.
- **Sybil attacks:** Attackers can create multiple fake identities to control a large portion of the oracles, allowing them to manipulate the data that is fed into the blockchain.
- **Bribing oracles:** Attackers can bribe oracles to feed false information into the blockchain, potentially causing the smart contract to make incorrect decisions.
- **Oracle censorship:** Attackers can prevent oracles from accessing certain data sources, leading to a lack of information for the smart contract to make decisions.
- **DoS:** Attackers can overload oracles with too many requests, making them unavailable to the smart contract and potentially causing it to fail.
- **Freeloading attacks:** In this type of attack, a node can utilize another oracle or off-chain component, such as an API, and simply replicate the values without any validation. For instance, an oracle responsible for providing weather data may expect data providers to measure temperature at a designated location. However, nodes may be incentivized to use a publicly accessible weather data API and simply present their data to the system. This can lead to centralization issues with the data source and, if carried out on a large scale, can significantly affect the accuracy of the data. This is particularly noticeable when sampling rates vary, for instance, when the on-chain oracle expects a 10-minute sampling rate while nodes that are freeloading provide data from an API that is updated only once per hour. The prevalence of freeloading in decentralized oracle data marketplaces can exacerbate a downward price spiral, as freeloading requires only a simple data retrieval.

Attacks on wallets

Cryptocurrency wallets can be subject to several attacks, listed below:

- **Phishing attacks:** Where an attacker creates a fake wallet website or app that looks similar to a real one, tricking users into revealing their seed phrase or private keys.
- **Malware attacks:** Where malware infects the computer or mobile device used to access the cryptocurrency wallet and steals seed phrases or private keys.
- **Man-in-the-Middle (MITM) attacks:** Where an attacker intercepts and manipulates transactions by posing as a trusted entity during the communication between the wallet and the Ethereum network.

- **Security vulnerabilities in the cryptocurrency wallet code:** Where a hacker exploits a weakness in the wallet code to gain access to private keys and consequently the funds.

Hardware wallets can be subject to some attacks, listed below:

- **Supply chain attacks:** Where an attacker modifies the hardware wallet during production or distribution to gain access to private keys or seed phrases.
- **Physical tampering:** Where an attacker physically opens the hardware wallet and accesses the private keys or seed phrases stored within.
- **Malware attacks:** Where malware infects the computer used to connect to the hardware wallet and steals private keys or seed phrases.
- **MITM attacks:** Where an attacker intercepts and manipulates transactions by posing as a trusted entity during the communication between the hardware wallet and the computer.
- **User-level attacks:** If users do not follow instructions correctly or adopt good practices to secure their keys, then this could simply lead to a loss of funds. There are many examples from the wild ranging from lost hard disks containing Bitcoin wallets to forgetting passwords to wallets and even forgetting pin codes to hardware wallets.

Frontend attacks can include several types:

- **Malicious scripts:** Usual scripting attacks on a web page to steal keys or account details or create rogue transactions
- **Misaligned frontend:** WFE functionality/UI not matching the backend contract
- **Account hacking:** Account details phished/lost for the WFE
- **DoS attacks:** WFE SPOF

At the interface layer, it is possible for cryptojacking attacks to be carried out against browsers and the operating system.



Cryptojacking is the unauthorized use of someone else's computer to mine cryptocurrency. It is usually done without the knowledge or consent of the computer owner and can slow down the affected machine as it uses its resources to mine the cryptocurrency. The mining process involves solving complex mathematical puzzles to validate transactions on the cryptocurrency's network, and it requires a lot of computing power. Cryptojackers can use malware to infect a computer and then use its resources to mine cryptocurrency, or they can use browser-based scripts that run on a website and mine cryptocurrency using the resources of the visitor's computer. There are several ways to remediate cryptojacking. First, ensure up-to-date antivirus software is running on the host. Chrome extensions such as no coin or miner block can defend against cryptojacking attacks. Another technique is to null route the mining URIs by updating the host's file or DNS settings. Also, use adblocker extensions, which can stop the cryptojacking script from running.

So far, we've discussed attacks on layer 1 blockchains. However, with the advent of layer 2 blockchains, new classes of attacks have emerged, which we discuss next.

Attacks on layer 2 blockchains

Some of the attacks are similar to layer 1 attacks; however, some are novel and specific to layer 2:

- **Transaction censoring:** In layer 2 systems, if a rollup provider is centralized, that could result in transaction censoring.
- **Attacks on a rollup provider:** There can be several attacks against a rollup provider, including DoS, and any other usual attacks that could be carried out against a node, including viruses/malware and network attacks.
- **Data availability attacks:** There are two options available for data availability. One is to store the data on a chain to ensure its availability and the other is to store it off-chain with third-party entities, which will make it available when required. Usually, economic incentives are applied to reward off-chain entities to store data. However, badly designed incentive mechanisms or centralized service providers could censor the data, which can lead to unintended consequences. Also, in the case of storing data on a layer 1 chain, the efficiency and benefits of rollups would be limited.



Data availability is required to make the committed data available to any party outside the rollup provider (layer 2) to ensure liveness if the rollup provider fails or disappears for whatever reason (even malicious). Data availability is also required if some user wants to withdraw their funds and need to bypass the layer 2 entirely, either due to unavailability of layer 2, or malicious actors on layer 2, or simply “want their money back.”

- **Blockchain bridge-related vulnerabilities:** This is a very common problem (as of early 2023). Several high-profile attacks like the Ronin Bridge attack, Wormhole attack, BNB smart chain attack, and Nomad Bridge have resulted in the loss of over a billion US dollars. Cross-chain bridges are very desirable features in the blockchain ecosystem as they enable cross-chain communication, which enables interoperability, results in better liquidity and provides services like cross-chain token transfers. However, there are several issues that can result in vulnerabilities. For example:
 - A badly designed bridge, if exploited, can lead to the loss of funds.
 - Also, the design of the bridges is usually quite complex, which can also lead to vulnerabilities that can be exploited by hackers.
 - There are several exploits and vulnerabilities, such as false deposits, validator design flaws, and validator takeover by malicious actors. Usually, these bridges work on the basis of events generated from one chain and read by the other chain to generate tokens or initiate a transfer on the other chain. If an attack somehow can produce a valid event without actually depositing funds on one chain (a false deposit), then it can result in generating new coins on the other chain, without really spending any coins on the first chain.
 - If the validators misbehave, collude, or are taken over by a malicious entity (adversary), then this can also lead to unintended consequences.

- **State channels and side chain-related attacks:** Attacks against state channels include channel jamming and replay attacks. Channel jamming is particularly problematic on a lightning network. It occurs when a DoS attack prevents nodes from routing while their liquidity is locked up and they are unable to earn fees because they cannot forward payments. An attacker can route the payment via other nodes under their control and refuse to finalize the payment. This situation will result in effectively “jamming the channel” until the HTLC time lock expires, and the payment is refunded.

Side chain attacks can include:

- Centralized block producers, which can lead to transaction censorship.
 - No inherited security from layer 1 (independent layer 2 security) can lead to stations where a side chain tries to run its own validator set and security mechanism, which could be at a lot lower level than layer 1. This can result in security issues and exploitation vulnerabilities on side chains.
 - No atomicity between two chains.
 - Vulnerable bridge.
- **DSL bugs:** As more and more **Domain-Specific Languages (DSLs)** like ZoKrates, Circom, and Cairo are being written to facilitate the creation of zero-knowledge proofs to be used in a layer 2 system, if they are not designed and/or implemented correctly, it could lead to language-level bugs, which can make the system vulnerable to certain attacks.

Cryptography layer

Several vulnerabilities and attacks can exist in the cryptography used at each layer:

Attacking public key cryptography

Public key cryptography is a crucial aspect of blockchain security and, generally, information security, but it is not 100% immune to attacks. Some common attacks against public key cryptography are listed below:

- **Brute-force attack:** A brute-force attack involves trying every possible combination of characters to find the private key. This attack can be prevented by using longer keys, but it becomes infeasible as the length of the key increases.
- **MITM attack:** In a man-in-the-middle attack, the attacker intercepts communication between two parties and replaces the public key of one party with their own public key. The attacker can then decrypt and read all messages sent between the two parties. A common example of a man-in-the-middle attack is *SSL stripping*, where the attacker intercepts communication between a client and a server and downgrades the connection to an insecure version, such as HTTP instead of HTTPS.
- **Impersonation attack:** In an impersonation attack, an attacker creates a fake certificate that appears to be from a trusted source, in order to trick a user into accepting the attacker’s public key. Phishing attacks can be used to impersonate trusted websites and steal sensitive information.

This is particularly a problem in the blockchain ecosystem where users could be victims of phishing attacks or poor security of the web frontends for DAPPs make them vulnerable.

- **Key reuse attack:** A key reuse attack occurs when the same key is used for more than one purpose, making it easier for an attacker to gain access to the private key. For example, the **Wired Equivalent Privacy (WEP)** encryption protocol used in early Wi-Fi networks was vulnerable to key reuse attacks, as the same key was used to encrypt all packets. In blockchain networks, it's advisable to use a key only for the purpose for which it was generated.
- **Same key used for a long time:** As a good practice, if a private key has served its purpose, e.g., for signing a blockchain transaction, then it's advisable to use a new key for a new transaction, instead of reusing the same key again and again.
- **Side-channel attack:** A side-channel attack exploits information leaked from the implementation of a cryptographic algorithm, such as power consumption, electromagnetic radiation, or timing information, to obtain information about the secret key. For example, **Differential Power Analysis (DPA)** is a type of side-channel attack that analyzes the power consumption of a device to extract the secret key used in encryption.

Such attacks can be mitigated by using strong encryption algorithms and proper key management practices and verifying the authenticity of the public keys used in communication.

Attacking hash functions

Several common attacks that can be carried out against hash functions are as follows:

- **Collision attack:** A collision attack is an attempt to find two inputs that produce the same hash output, also known as a hash collision. A successful collision attack can compromise the integrity of digital signatures and other applications that rely on hash functions.
- **Preimage attack:** A preimage attack is an attempt to find an input that hashes to a specific output value. A successful preimage attack can compromise the security of password storage systems and other applications that use hash functions to store secrets.
- **Birthday attack:** A birthday attack is a type of collision attack that exploits the birthday paradox to find collisions more efficiently than a brute-force search. This type of attack is especially relevant for hash functions with smaller outputs, such as MD5 and SHA-1.
- **Length extension attack:** A length extension attack is a type of attack that allows an attacker to append data to a message after it has been hashed, without knowing the original message or hash value. This type of attack is possible when using hash functions with certain properties, such as the Merkle-Damgård construction.

In blockchain systems, usually SHA-256 and SHA-3 and Keccak are used, which are considered secure against these threats, but with a quantum computer, it might be possible to find some collisions.

Key management-related vulnerabilities and attacks

Key management-related vulnerabilities and attacks can compromise the confidentiality, integrity, and availability of encrypted data and result in significant financial and reputational damage. Some key management-related vulnerabilities and attacks are listed below:

- **Insecure key storage:** Insecure key storage can result in the exposure of private keys, making it easier for an attacker to sign illegitimate transactions. To mitigate this vulnerability, private keys should be stored in a secure location, such as a **Hardware Security Module (HSM)** or encrypted file, and access should be restricted to authorized personnel. It is also advisable to use a hardware wallet such as Trezor, SafePal, etc. to secure the private keys. User crypto wallets must be sure to use good cryptography and security practices to mitigate this threat.
- **Unauthorized key sharing:** Unauthorized key sharing can result in the exposure of private keys, making it easier for an attacker to gain access to encrypted data. To mitigate this vulnerability, access to private keys should be restricted to authorized personnel and policies should be in place to prevent unauthorized sharing.
- **Key loss or theft:** Key loss or theft can result in the permanent loss of access to encrypted data. To mitigate this vulnerability, keys should be stored in multiple locations and backups should be made to ensure that keys can be recovered in the event of loss or theft.
- **Key escrow attack:** Key escrow is a process in which a third party holds the private keys for encrypted data. Key escrow attacks can occur when the third party holding the keys is compromised, making it easier for an attacker to gain access to encrypted data. To mitigate this vulnerability, keys should be stored in a secure location and access should be restricted to authorized personnel.

ZKP-related attacks

There can be several attacks that could target zero-knowledge proofs:

- **Inadequate bit security:** If an inadequate bit length is chosen for security, that could result in a security compromise; e.g., if instead of choosing 4,096-bit security in RSA, only 512 bits are used, which would result in it being exploited, as 512-bit is easy to break. Similarly, in the context of layer 2, some claims have been made, for example, by StarkWare, that SHARP prover runs at 80 bits of security. However, the FRI commitment scheme only runs at 48 bits of security with very lax assumptions about its soundness. The formally proven security turns out to be at most 22 bits. The mechanism also use a 32-bit PoW puzzle to mitigate this limitation and achieve a higher security level, but this limitation could still be exploited. This means that in practice, the probability of forging a proof could be higher than expected.
- **Attacks on privacy:** There could be several attacks against privacy, including deanonymization attacks, confidentiality breaches, and identity disclosure attacks.
- **Digital signature vulnerabilities:** Digital signature malleability refers to the ability to modify a digital signature in a way that does not invalidate it but still modifies the underlying message. This vulnerability can have serious consequences for the security of digital signatures, as it can lead to undetected tampering with signed messages and the circumvention of cryptographic protocols.
- **Quantum threats:** It is possible, with the availability of quantum computers, for some schemes like ECC and RSA to be broken. Quantum computers pose a threat to the security of blockchain technology because they have the ability to break the cryptographic algorithms that are currently used to secure transactions on most blockchain networks.

For example, the popular public-key cryptography system RSA and **Elliptic Curve Digital Signature Algorithm (ECDSA)** can be broken by a large enough quantum computer, making it possible to compromise the security of private keys and steal cryptocurrencies. To mitigate the quantum threats against cryptography, several solutions are being developed, including:

- **Post-quantum cryptography:** This involves the development of new cryptographic algorithms that are quantum-resistant, meaning they cannot be broken by quantum computers.
 - **Quantum-safe signature schemes:** This involves modifying existing signature schemes to make them quantum-resistant, such as the development of hash-based signature schemes, such as Lamport signatures and Winternitz signatures.
 - **Quantum Key Distribution (QKD):** This is a secure communication method that allows two parties to establish a shared secret key by transmitting quantum information, which is immune to eavesdropping.
 - **Hybrid approaches:** This involves combining classical cryptographic algorithms with quantum-resistant algorithms to enhance security against quantum attacks.
- **Proof malleability:** Getting the cryptography right is quite tricky, which is especially true with zk-SNARKs, which is a somewhat new subject. Problems with the underlying mathematics and then implementation can lead to unintended consequences. One of the issues is proof malleability where an attacker can alter a proof without invalidating it. In other words the attacker can manipulate the proof in a way that it still appears to be valid and authentic, but its content has been altered. This can lead to various security issues, such as replay attacks, where same proof can be used multiple times for different messages.
 - **Setup vulnerabilities:** If the trusted setup is not conducted properly or is compromised and secret values leak, then it can allow a prover to create false statements. It is important to delete the secret values in the trusted setup, i.e., so-called toxic waste, forever.

Security analysis tools and mechanism

There are several techniques to check the correctness of programs. First, we have unit testing, which is a common method. Second, we have property-based testing, i.e., fuzzing. At the next level, we have model checking, and finally, formal proofs are the most advanced technique to ensure the correctness of programs, in this case, smart contracts. Testing includes unit tests, integration tests, full end-to-end tests, and property-based testing, also called fuzzers.

Static analysis allows us to check the code against a set of coding rules to find code defects. The code does not execute; instead, it is statically checked. On the other hand, there is a dynamic analysis technique where the code is executed to find bugs and is tested against test criteria. Dynamic analysis usually constitutes unit tests and is not considered a formal verification technique. Static analysis using formal techniques is used to formally verify the correctness of the program.

There are three types of proofs. We are familiar with pen-and-paper proofs, which are written by hand and verified manually. This can be prone to errors. Secondly, we have proof assistants, which provide support while writing proofs.

Proofs are still written manually by a human, but the proof can be checked for correctness automatically. Also, the tool provides support in writing the correct proof. Finally, we have automated proofs, but this suffers from an undecidability problem and exponential search space. It's worth noting that it takes a lot of effort to build formal proofs, so it only makes sense to use a formal verification approach for code and algorithms that are complex and where correctness is hard to establish. Usually distributed systems, concurrent systems, and safety-critical systems require formal verification. For straightforward code, it is not worth spending time and effort on building formal proofs.

For smart contract formal verification, some common tools include VERX and KEVM. The formal verification of smart contracts can allow the detection of complex bugs, which are hard to detect manually, or using simple automated tools or unit testing. The formal verification of smart contracts proves that the smart contract satisfies a formal specification of its functionality. Manual analysis, while a cumbersome process, has its place too to ensure the correctness of smart contract code. Auditors can manually go through code to try and find bugs and vulnerabilities. However, this method is not very effective and is very time-consuming.

While the usual testing of code with automated tools, where unit tests are written, is usually an acceptable technique, code coverage is not a hundred percent guaranteed in some cases, leading to bugs in the software. Manual analyses and, in fact, most methods suffer from false negatives and false positives. False negatives can be defined as the inability to find vulnerabilities that actually existed in the code. False positives can be defined as the report of the existence of vulnerabilities that are actually not vulnerabilities. Symbolic execution is a technique for checking program correctness. It uses symbolic inputs to represent a set of states and transitions formulated in a precise mathematical language.

Formal verification has become a very interesting topic in the blockchain space, due to its strong technical merits as a technique to ascertain the correctness of code, algorithms, and protocols related to blockchain. Note that formal verification is not a new technique, but due to the strict security requirements of blockchain systems, it has found suitable applications in blockchain technology.

Formal verification

Before diving into different formal verification techniques that are available in the blockchain space, first, let's develop some understanding of what **formal verification** is, what its types are, and why it is desirable.

Formal methods are the set of techniques used to model systems as mathematical objects. These methods include writing specifications in formal logic, model checking, verification, and formal proofs. Generally, formal methods can be divided into two broad disciplines called **formal specifications** and **formal verification**. The first discipline is concerned with writing precise and concrete specifications, whereas the latter encompasses the development of proofs to prove the correctness of a specification.

In essence, formal verification is comprised of three steps:

1. First, create a formal model of the system to be checked.
2. Second, write a formal specification of the properties that are expected to be satisfied by our model.
3. Finally, the model is checked to ensure that the model satisfies the specification.

For checking, there are two broad categories of techniques that can be used, namely *state exploration-based* approaches and *proof-based* approaches. There are pros and cons to both:

- State exploration-based approaches, where all possible states are checked, are automatic but are inefficient and difficult to scale. A usual problem is state explosion, where the number of states to check grows so exponentially big that the model does not fit in a computer's memory.
- On the other hand, proof-based approaches (theorem proving) are more precise but are somewhat challenging to implement because they require manual proof writing by a human. Even though these proofs are assisted by the proof assistant, they require more in-depth knowledge of the proofs and more time and effort to implement.



Proof-based verification is performed using proof assistants such as Coq (<https://coq.inria.fr/>) and Isabelle (<https://isabelle.in.tum.de/>).

Formal verification of smart contracts

A lot of research on the formal verification of smart contracts has also been conducted. The proposed techniques include, but are not limited to, model checking, static analysis, dynamic analysis, and verification using proof assistants.

Smart contract security is a very exciting and vast area of research. It is almost impossible to cover all the aspects in this short chapter. You are encouraged to read the introduction to formal verification earlier in this chapter and then go through some of the preceding papers for further research.

After this discussion on smart contract security, we should now understand its significance. While all the issues and techniques discussed here to mitigate the security issues in smart contracts are valuable, a question still arises about what else we can do, given that smart contract security is a sensitive topic that can result in serious financial losses.

The answer could be that we treat smart contracts and the underlying blockchain as safety-critical systems, and then apply all attributes of safety-critical systems to smart contracts. From an engineering perspective, it will become a subject of safety-critical systems and will be treated as such. This idea might sound a bit over the top, but imagine if, one day, a smart contract is responsible for generating an event that would trigger a shutdown as a result of overheating in a nuclear reactor. It's only logical that every safety precaution is taken. Again, a little exaggerated perhaps, but entirely possible. With IoT and blockchain convergence, such scenarios might become a reality soon, where nuclear reactors are running using a blockchain as a mechanism to control and track associated operations. Similar critical scenarios can be found in healthcare, aviation, and defense.

In Solidity, formal verification is achieved through the use of **Satisfiability Modulo Theories (SMT)** and **Horn solving**, where the `SMTChecker` module analyzes the code to ensure that it meets the conditions outlined in the `require` and `assert` statements. If an error is found, the `SMTChecker` provides a counterexample to the user. If no errors are found, it is assumed that the code is safe. The `SMTChecker` also checks for potential issues such as arithmetic overflow and underflow, division by zero, unreachable code, access to an out-of-bounds index, insufficient funds for a transfer, and more.

We just introduced two new terms, SMT and Horn solving. Let's see what they are.

SMT is a well-established decision procedure that assesses the satisfiability of logical formulas in the presence of predefined theories. This technique is widely used in the field of formal verification and automated theorem-proving for evaluating the correctness of various systems. The SMT solver processes a logical formula as input and returns a binary outcome of either "SAT" or "UNSAT" to indicate whether the formula can be satisfied given the predefined theories. The application of SMT has been shown to provide significant benefits in the development of high-quality systems, as it enables the automatic proof of desired properties and the detection of potential bugs and errors before they occur.

Horn solving is a technique used in formal verification to automatically prove the correctness of a program or system. It is based on the notion of Horn clauses, which are a type of logical statement made up of one or more literals and at most one positive literal. In formal verification, the Horn solver is used to check if the program satisfies a set of Horn clauses, which represent the desired properties of the system. Horn solving works by reducing the proof obligation to a set of Horn clauses and then using a theorem prover to check if there is a solution that satisfies all the clauses. The theorem prover applies a combination of decision procedures and search algorithms to find a solution. If a solution is found, it is proof that the program meets the desired properties. If no solution is found, the Horn solver produces a counterexample, which is a trace of execution that violates one of the properties.

In the next section, we'll introduce a formal verification technique called model checking, which is used to check a system model for its correctness.

Model checking

Model checking can be defined as a technique used to verify finite state systems automatically. The underlying process is based on running an exhaustive search of the state space of the system. With this brute-force approach, it can be determined whether a specification is true or false. This option is becoming more popular because it does not require time-consuming proof writing.

Instead, this paradigm allows a designer to write the specification and its properties formally and the model checker will automatically explore the entire state space and evaluate the model against the specified properties. For example, if the specification says that a particular state should never be reached, and during state exploration, the model checker finds that there is an execution (trace) that does enter that very state that is undesirable, then it will report that. The designer can then address the problem accordingly.

A visual representation of the model checking process is shown in the following diagram:

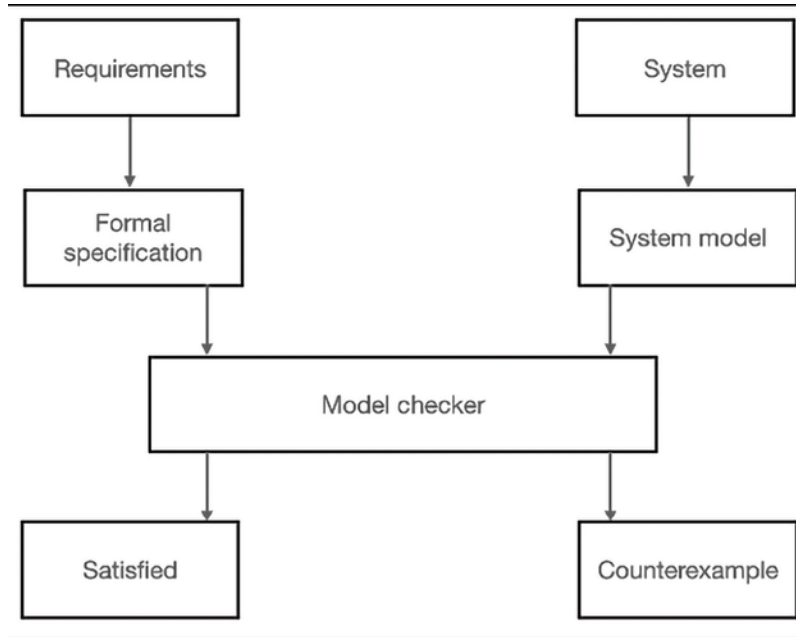


Figure 19.2: Model checking

The process of model checking starts with modeling, where a formal specification is created from an informal design of the model. The specification step comes next, where relevant properties of the design are specified using some logical formalism. Finally, verification of the model is carried out, which checks all the properties defined in the specification and provides results. In the case of false or negative outcomes, error traces with counterexamples (roughly defined as exceptions) are provided to help the designer trace the error.

Mathematical assertion languages called temporal logics are used to describe the states of a system over time. Temporal logic is used at the specification step of the model checking process, which represents the evolution of the behavior of a system over time, i.e., the ordering of events of a system over time. In this paradigm, a specification of the behaviors of the system over time is built using various fundamental properties of individual states. For example, in a state, it can be said that “something is on” or “something is active.” There are also propositional connectors used in the specification design, such as *AND* \wedge , *OR* \vee , *NOT* \neg , and *IMPLY* \Rightarrow .

Finally, there are also temporal connectives, which state something like the system always only has one process active, or the system can never be in a state where both read and write operations are performed simultaneously.

There are two types of temporal logic, namely **Linear Temporal Logic (LTL)** and **Computation Tree Logic (CTL)**. LTL is concerned with the model properties on a single computation path or execution, whereas CTL is used to express properties over a tree of all possible paths.

There are a few fundamental temporal operators used in LTL to describe a system's behavior. These properties can be used to specify the safety and liveness properties of a system. The formulas are $\Box p$ (always), $\Diamond p$ (eventually), U (until), and \bigcirc (next). The always and eventually properties are most used.

For example, $\Box ((\text{message}) \neg \text{signed} \vee \neg \text{sealed}) \Rightarrow \bigcirc \neg \text{valid}$ means that it is always true that if a message is not signed or sealed, then the message is not valid. Another example could be $\Box ((\text{broadcast}) \Rightarrow \Diamond \text{received})$ with broadcast implying that eventually (sometimes referred to as possible or in some future time) the message will be received. Another general example could be that of a microwave oven ensuring that heating must not start until the door is closed, which can be represented in temporal logic as $\neg \text{StartHeating } U \text{ Door}(\text{closed})$.

Formal methods are needed to ensure the safety of a system and have been in use in the avionics, electronics, hardware, and embedded systems industries for quite some time, as well as most recently in the software industry. Due to the renewed interest in distributed systems research with the advent of blockchain, and several high-profile incidents resulting in financial loss, such as the infamous **DAO hack**, the need to use formal methods in the blockchain space is becoming increasingly prominent. Some initiatives have already been implemented; however, there is still a long way to go.

Blockchain needs to leverage the research that has already been done in the distributed systems and formal verification space, so that not only can existing blockchain systems be made safer but new platforms can also be developed based on a formal specification. It should be noted that not all aspects of a blockchain need to, or should, be formally verified, due to the time-consuming and highly involved nature of the verification process. Therefore, at times, it is sufficient to model and test the most critical parts of a blockchain, such as consensus and security protocols.

There are many dimensions in a blockchain where model checking and specification using temporal logic can be quite useful. It can be used to describe an aspect of the blockchain platform formally, which can then be verified formally to ensure that all required properties are met. For example, in smart contract development, regardless of the language used, a model could be developed and checked before the actual coding of the smart contract. This approach will ensure that the program code, if programmed correctly according to the verified specification, will behave as it is intended. As smart contracts are used in all kinds of critical use cases, including financial transactions, it is advisable to apply model checking in this domain to ensure the fairness and security of the system.

Now, we'll briefly look at how consensus mechanisms can be verified.

Verifying consensus mechanisms

From another angle, a **consensus mechanism**, such as **PBFT**, exists in a blockchain, which ensures that all the nodes in a blockchain network reach an agreement on the proposed values, even in the presence of faulty nodes. Consensus mechanisms such as PBFT and Raft were described in *Chapter 5, Consensus Algorithms*.

This is an area of prime importance and due to its critical nature is regarded as the most vital core mechanism of a blockchain. Model checking can play a crucial role in the formal description and verification of consensus algorithms, to ensure that the protocols meet the required security properties.

A general approach to verifying the properties of a consensus algorithm in blockchain starts with specifying the requirements, required properties, and specification in a formal language.

There are several options available for modeling and verifying programs, but tools such as TLA+ and the TLC model checker are making the processes more accessible and are becoming more prominent in this space. Another popular tool is known as SPIN, which uses PROMELA to write specifications. However, note that there is no single right answer: the choice of formal verification tools mostly depends on the judgment and experience of the designer, the type and depth of verification required, and, to a certain degree, the usability of the verification tools.

A distributed consensus algorithm is evaluated against two categories of correctness properties, namely *safety* and *liveness*. Safety generally means that nothing bad will happen, whereas liveness implies that something good will eventually occur. Both of these properties, then, have some sub-properties, depending on the requirements. Usually, for consensus mechanisms, we have *agreement*, *integrity*, and *validity* attributes under safety properties and for liveness, often, *termination* is a desired property.

We can say that a program is correct if, in all possible executions, the program behaves correctly according to the specification. The specification is formally defined, which is then verified using a model checker or theorem provers.

With this, we've completed our basic introduction to formal verification. Now, let's have a look at smart contract security and see what formal tools are available for smart contract security verification.

Smart contract security

Recently, a lot of work has been started relating to smart contract security and, especially, the formal verification of smart contracts is being discussed and researched. This was all triggered especially by the infamous DAO hack and other attacks.

Formal verification is a process of verifying a computer program to ensure that it satisfies certain formal statements. This is not a new concept and there are a number of tools available for other languages that achieve this; for example, Frama-C (<https://frama-c.com>) is available for analyzing C programs.

The key idea behind formal verification is to convert the source program into a set of mathematical statements that is understandable by the automated provers. For this purpose, **Why3**, which is a platform for program verification, is commonly used.



Note that an experimental but operational Why3 verifier was available in Remix IDE initially but was later removed. However, now, in Remix IDE, static analysis options are available. Details can be found here: https://remix-ide.readthedocs.io/en/latest/static_analysis.html.

Smart contract security is of paramount importance now, and many other initiatives have also been taken in order to devise methods that can analyze Solidity programs and find bugs. Some examples include **Oyente**, **Manticore**, and **Slither**, which are the tools built by researchers to analyze smart contracts.

Smart contracts and, generally, all aspects of blockchain can be formally verified. Static analysis of Solidity code is also now available as a feature in the Solidity online Remix IDE.

The code is analyzed for vulnerabilities and reported in the **Analysis** tab of Remix IDE. Static analysis in Remix IDE analyzes several categories of vulnerabilities, including **Security** and **Gas & Economy**. Other than static analysis capabilities within Remix IDE, other tools are also available, which we'll discuss next:

- Slither is a static analyzer framework for smart contracts.
- Manticore is a **symbolic execution** framework that can be used for binaries and smart contracts. More information on Manticore can be found here: <https://arxiv.org/pdf/1907.03890>.

Symbolic execution is a method used to analyze computer programs to determine which part of the computer program executes as a result of what input. In other words, it establishes which input executes which part of the program. In symbolic execution, instead of actual input data, symbolic values are used, which are then used to evaluate the program. Also, an automated theorem prover is used to check whether there are values that result in incorrect behavior of the program or cause it to fail. It is used for debugging, software testing, and security.

Oyente

Currently, **Oyente** is available as a Docker image for easy testing and installation. It is available at <https://github.com/melonproject/oyente> and can be downloaded and tested. In the following example, a simple contract taken from the **Solidity** documentation that contains a reentrancy bug has been tested. First, we will show the code with a reentrancy bug:

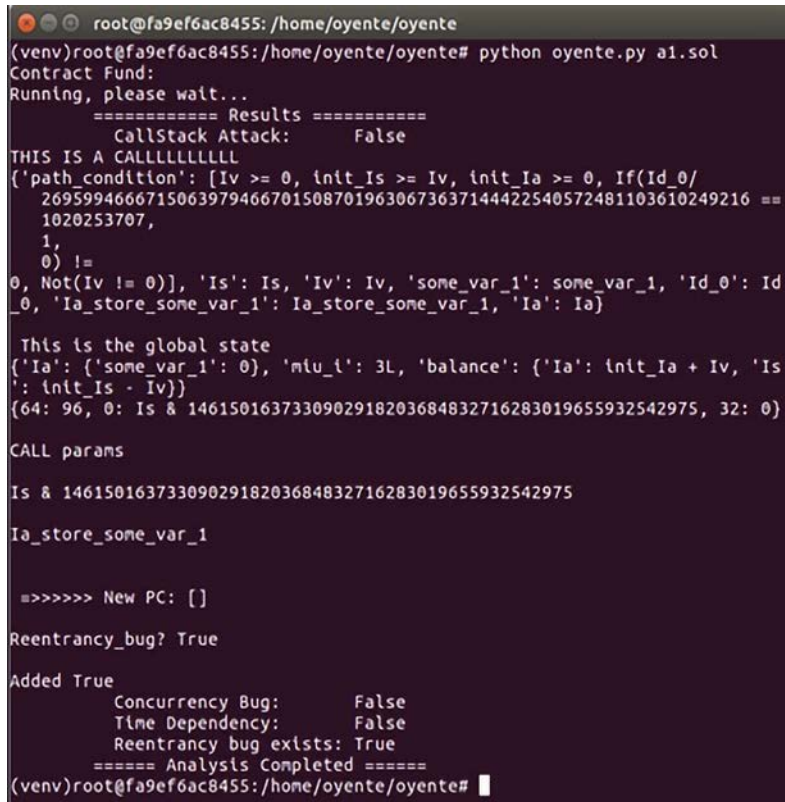
```
1 pragma solidity ^0.4.0;
2 contract Fund {
3     mapping(address => uint) shares;
4     function withdraw() public {
5         if (msg.sender.call.value(shares[msg.sender]))()
6             shares[msg.sender] = 0;
7     }
8 }
9 }
```

Figure 19.3: Contract with a reentrancy bug, sourced from the Solidity documentation

This sample code contains a reentrancy bug, which basically means that if a contract is interacting with another contract or transferring ether, it is effectively handing over the control to that other contract. This allows the called contract to call back into the function of the contract from which it has been called, without waiting for completion. For example, this bug can allow calling back into the `withdraw()` function shown in the preceding example again and again, resulting in obtaining ETH multiple times. This is possible because the share value is not set to 0 until the end of the function, which means that any later invocations will be successful, resulting in withdrawing again and again.

An example is shown of Oyente running to analyze the contract shown here, and as can be seen in the following output, the analysis has successfully found the reentrancy bug.

The bug is proposed to be handled by the Checks-Effects-Interactions pattern described in the Solidity documentation:



```

root@fa9ef6ac8455: /home/oyente/oyente
(venv)root@fa9ef6ac8455:/home/oyente/oyente# python oyente.py a1.sol
Contract Fund:
Running, please wait...
===== Results =====
CallStack Attack:      False
THIS IS A CALLLLLLLLLL
{'path_condition': [Iv >= 0, init_Is >= Iv, init_Ia >= 0, If(Id 0/
26959946667150639794667015087019630673637144422540572481103610249216 ==
1020253707,
1,
0) !=
0, Not(Iv != 0)], 'Is': Is, 'Iv': Iv, 'some_var_1': some_var_1, 'Id_0': Id
_0, 'Ia_store_some_var_1': Ia_store_some_var_1, 'Ia': Ia]}

This is the global state
{'Ia': {'some_var_1': 0}, 'mlu_t': 3L, 'balance': {'Ia': init_Ia + Iv, 'Is
': init_Is - Iv}}
(64: 96, 0: Is & 1461501637330902918203684832716283019655932542975, 32: 0)

CALL params

Is & 1461501637330902918203684832716283019655932542975

Ia_store_some_var_1

=>>>>>> New PC: []

Reentrancy_bug? True

Added True
Concurrency Bug:      False
Time Dependency:     False
Reentrancy bug exists: True
===== Analysis Completed =====
(venv)root@fa9ef6ac8455:/home/oyente/oyente#

```

Figure 19.4: Oyente tool detecting Solidity bugs

With this example, we conclude our introduction to security and analysis tools for Solidity. This is a very rich area of research, and more and more tools are expected to be available with time.

Solgraph

Visualizing the function control flow in a program makes it easy to understand the flow and analysis better. A tool called Solgraph can be used to generate a graph of the function control flow of a Solidity contract and finds possible security vulnerabilities. More information on this tool is available here: <https://github.com/raineorshine/solgraph>.

Threat modeling

Threat modeling is the process of identifying, analyzing, and documenting potential threats to a system or application. The goal of threat modeling is to identify vulnerabilities and potential attack vectors so that appropriate security controls can be put in place to mitigate the risk. Threat modeling is a standard practice carried out in traditional information security scenarios; however, the same techniques can be applied to blockchain as well.

There are many ways threat modeling can be performed for blockchain by using standard threat modeling techniques. There are several different standard models that can be used for threat modeling, including:

- **STRIDE:** This model, developed by Microsoft, helps identify and categorize threats based on the type of threat they pose. STRIDE stands for Spoofing, Tampering, Repudiation, Information Disclosure, DoS, and Elevation of Privilege.
- **DREAD:** This model, developed by the **Open Web Application Security Project (OWASP)**, helps assess the risk of a threat based on its likelihood of occurrence and the potential impact it could have. DREAD stands for Damage, Reproducibility, Exploitability, Affected Users, and Discoverability.
- **CVSS:** The **Common Vulnerability Scoring System (CVSS)** is a standardized method for rating the severity of vulnerabilities based on their impact and likelihood of exploitation.
- **Attack trees:** This model represents threats as a tree structure, with the root node representing the attack goal and the child nodes representing the various steps or actions required to achieve the goal.
- **Threat matrix:** This model is a visual representation of the likelihood and impact of potential threats. It allows analysts to prioritize threats based on their potential risk.

We'll describe STRIDE in a bit more detail now, and how it can be applied to blockchain. Note that other models can also be used and adapted for use in the blockchain ecosystem.

STRIDE is a security model that helps identify potential threats and vulnerabilities in a system or network. It stands for:

- **S – Spoofing:** This refers to the act of impersonating someone or something to gain access to a system or information.
- **T – Tampering:** This refers to the act of altering or manipulating data or information in a system.
- **R – Repudiation:** This refers to the ability to deny that an action or event took place.
- **I – Information Disclosure:** This refers to the act of disclosing sensitive or confidential information to unauthorized parties.
- **D – DoS:** This refers to an attack that is designed to make a system or network unavailable to legitimate users.
- **E – Elevation of Privilege:** This refers to the act of gaining unauthorized access to a system or information with higher privileges than one's own.

STRIDE helps identify and prevent these types of threats by examining the system or network and looking for ways to reduce the risk of these types of attacks. This can be done through a variety of methods, such as implementing strong authentication and access controls, encrypting sensitive data, and regularly updating and patching systems to fix vulnerabilities.

Benefits of using the STRIDE model include:

- It helps identify potential threats and vulnerabilities in a system or network.

- It provides a systematic approach to evaluating and addressing security risks.
- It can be used to develop a comprehensive security plan for a system or network.

To implement STRIDE, we would first need to identify the assets and resources in the blockchain ecosystem that need to be protected. We would then need to evaluate the potential threats and vulnerabilities to those assets and determine the appropriate measures to reduce the risk of those threats. This could include implementing good practices and implementing controls and technical practices to prevent and mitigate spoofing, tampering, repudiation, information disclosure, DoS, and elevation of privilege attacks.

For example, if we apply STRIDE to do threat modeling on the consensus layer, we can see how for each dimension of STRIDE, we can model the impact of the attacks:

- **S – Spoofing:** In a consensus protocol, it might be possible to introduce a byzantine node that appears to be a legitimate node (validator). The malicious validator node can take over and impersonate an honest node.
- **T – Tampering:** Collusion or sybil attacks in a consensus protocol could result in the blockchain being written by rogue blocks with transactions that are created by the attacker.
- **R – Repudiation:** It is possible, due to tampering, for the attacker to repudiate past transactions or, due to control over the network (by collusion), simply repudiate any previous transaction, resulting in the loss of funds.
- **I – Information disclosure:** As the attacker has influence over the network due to tampering and spoofing, it could be possible for the attacker to extract sensitive information from the blockchain.
- **D – DoS:** It could be possible, e.g., in a BFT protocol, for a “follow the leader” type of attack to be launched. In this case, the attacker predicts or finds out due to a simple leader election algorithm which node is likely to be the next proposer (leader) node. The attacker can “follow the leader” and keep launching DoS attacks or malware attacks against that single node that is expected to be the next leader or is already a leader. If the DoS attacks are successful, then there won’t be any new block proposals made to the network because every time a leader is elected, the attacker attacks it with DoS or even launches DoS against the next expected leader. This can consequently stop block generation on the network, resulting in DoS.
- **E – Elevation of Privilege:** It is possible, with a byzantine node introduced in the system, for the attacker to gain elevated control over the network and thus the blockchain state.



Note that all STRIDE dimensions aren’t always applicable to every attack or layer. Sometimes only, for example, S or R is applicable, but not the other dimensions.

With this, we complete our introduction to threat modeling. STRIDE and other threat modeling techniques can be used in many other scenarios on the blockchain ecosystem, at every layer.

Regulation and compliance

Regulatory compliance is another dimension of blockchain security. How do we ensure that the blockchain platform is GDPR-compliant? How do we ensure that the blockchain platform is compliant with the enterprise security policy? Such questions fall into the category of regulatory compliance-related security requirements. There are certain requirements around data localization, e.g., customer data may not leave a specific geographic location. In such scenarios, a globally replicated blockchain may not be the best solution. In those cases, restricted private transactions (introduced in *Chapter 16, Enterprise Blockchain*) could be a useful construct to follow. In other situations, the use of off-chain transaction managers located in specific geographic locations could be helpful. In certain cases, specific cryptographic protocols are mandated by organizations such as NIST. For example, specific curves are mandated in SP 800-186 by NIST for federal agencies, which means that for a client to be used in government settings, it must use the same cryptography as mandated by NIST for US government use. Adhering to standards also facilitates interoperability, better security, and faster development. The document can be found here: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186.pdf>.

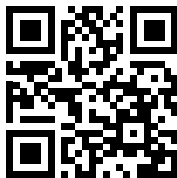
Blockchain can itself be used for malicious purposes. Blackhat hackers can use it as a vehicle to get paid for ransomware, for example, Petya, WannaCryptor, or Locky. There have been many ICO scams and Ponzi schemes. The point to note here is that blockchain is a tool that can be used for both good and bad. For example, Tornado Cash could be used for charitable purposes or money laundering. It is possible to follow some regulations to ensure the appropriate use of such platforms.

Summary

This chapter introduced blockchain security. We started by discussing a layered model of the blockchain ecosystem and its components. After this, we introduced attacks at different layers and how they can be mitigated. Moreover, we introduced the tools that can be used to analyze smart contracts and identify vulnerabilities. We also introduced STRIDE, which can be used to model threats in the blockchain ecosystem. In the next chapter, we will introduce identity, which is a major area of interest in the blockchain world. Note that blockchain security is a vast and deep subject and we cannot cover everything in a single chapter. However, this should provide a solid foundation to research further.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



<https://packt.link/ips2H>

